THE QUEEN'S AWARDS FOR ENTERPRISE 2001

BBC

R&D White Paper

WHP 112

May 2005

Hardware for arithmetic coding

P.J. Bleackley

Research & Development
BRITISH BROADCASTING CORPORATION

BBC Research & Development
White Paper WHP 112

Hardware for Arithmetic Coding

Peter J. Bleackley

Abstract

This white paper describes a fixed probability prototype of a VHDL implementation of an arithmetic coder for Dirac video coding. The technical background of arithmetic coding is also described.

Additional key words: video coding, Dirac, arithmetic coding, hardware

White Papers are distributed freely on request.

Authorisation of the Chief Scientist is required for publication.

BBC Research & Development
White Paper WHP 112

Hardware for Arithmetic Coding

Peter J. Bleackley

1 Introduction

Dirac is a general purpose video codec developed by BBC Research and Development, and released as open source so as to enable it to be used and modified freely by any interested parties. It uses motion compensation, wavelet coding, and arithmetic coding to obtain compression and image quality comparable with the best video codecs currently available.

Dirac has so far been implemented as software, written in C++. However, for many video applications it is desirable for a hardware implementation to be available. As a first stage to developing such hardware, an implementation of the arithmetic coding stage in VHDL is being developed. This white paper describes a fixed-precision prototype of this arithmetic coder.

Section 2 of this document describes the principles of arithmetic coding. Sections 3 and 4 describe the architecture of a fixed precision encoder and decoder respectively. Section 5 discusses the results of synthesizing this coder.

2 Arithmetic coding background

2.1 Theory

Consider a message consisting of a string of symbols. For a symbol s_k which occurs at a given position in the message with a probability p_k, then the information content of the symbol is $-\log_2(p_k)$. This is the minimum number of bits needed to encode the symbol. In general it will not be an integer.

Compression schemes such as Huffman coding, which encode each symbol with an integer number of bits, will in general only be optimal when $p_k = 2^{-q}$, where q is an integer, for all symbols. Otherwise there will be an overhead of typically 1/2 bit/symbol in encoding the message, as the number of bits in the encoded symbol will be rounded up to the nearest greater integer. To improve on this, it is necessary to encode the message in such a way that a given bit in the encoded stream may represent data from more than one symbol.

Consider the open interval $[0, 1)$. For an alphabet of symbols s_k, where $0 \leq k < n$, which occur with probabilities p_k

$$\sum_{k=0}^{n-1} p_k = 1$$

The interval may therefore be subdivided in proportion to the probabilities of the various symbols. Each symbol may therefore be represented by the interval $[c_{k-1}, c_k)$, where

$$c_k = \sum_{i=0}^{k} p_k$$

is the cumulative probability of all symbols up to and including s_k, and we define $c_{-1} = 0$. To represent a second symbol s_l, the interval may be further subdivided into the intervals $[c_{k-1} + p_k c_{l-1}, c_{k-1} + p_k c_l)$. The length of this interval is $p_k p_l$. This process may be carried out recursively for further symbols, and so a message of m symbols may be represented by an interval of length

$$\prod_{i=0}^{m-1} p_i$$

The number of bits required to specify that a point lies within this interval is then

$$H = -log_2 \left(\prod_{i=0}^{m-1} p_i \right) = - \sum_{i=0}^{m-1} \log_2 p_i$$

H is therefore the sum of the information contents of all the symbols in the message, and thus the information content of the message. A H bit number (rounded up to the nearest integer) representing a point in the interval will therefore be an optimal representation of the message. The process of identifying the number which represents the message in this way is called *arithmetic coding* [1].

2.2 Implementation

Certain practical considerations must be taken into account in a practical realisation of arithmetic coding. These are the need to use finite precision arithmetic, the need to ensure that underflows and overflows do not occur in the arithmetic, and the need to ensure that the decoder terminates at the correct point.

2.2.1 Finite precision

The discussion above implicitly assumed that the number representing the message could be calculated to as many bits as necessary. In general, the device calculating the arithmetic code will have only finite precision registers available to it. It must therefore be able to calculate a small portion of the code at a time, outputting data when it will no longer be affected by further calculations.

This can be achieved by the following mechanism. Consider a coder with two 16 bit registers, LOW and HIGH. LOW is initially set to 0000000000000000, representing 0, and HIGH is set to 1111111111111111, representing $1 - \delta$, where δ is in theory infinitesimal. When a symbol s_k is received by the encoder, the following transformations are carried out.

$$\text{LOW}' = \text{LOW} + c_{k-1} \times (\text{HIGH} - \text{LOW} + 0000000000000001)$$

$$\text{HIGH}' = \text{LOW} + c_k \times (\text{HIGH} - \text{LOW} + 0000000000000001) - 0000000000000001$$

(0000000000000001 is added to the difference between the registers and subtracted from the HIGH result because the interval must be treated as an open interval). After this, the most significant bits of the two registers are compared. If they match, this value is output and the remaining bits are shifted up one place. The least significant bit of LOW is set to zero and that of HIGH is set to one. This continues recursively until the most significant bits of the two registers no longer match.

Bits received by the decoder are stored in a 16 bit register, CURRENT. If the value stored in this register lies within the interval $c_{k-1} \leq \text{CURRENT} < c_k$, the symbol s_k is output and the high

and low registers are rescaled as for the encoding this symbol. If the most significant bits of HIGH and LOW match, the most significant bits of all the registers are discarded, and the remaining bits are shifted up one place. The least significant bit of HIGH is set to 1, that of LOW is set to zero, and that of CURRENT with data from the incoming stream.

This procedure may be explained as follows. The interval representing the message after a given number of symbols have been encoded always lies within the interval representing it before the most recent symbol was encoded. Thus, when this interval lies entirely within one half of the interval $[0, 1)$, the other half of the interval need no longer be considered and may be discarded. This allows the remaining half-interval to be considered with greater precision. There is some coding overhead from this approach, due to rounding errors, but it is small.

This mechanism has the further advantage that it allows encoding and decoding to be carried out in real time, whereas any mechanism that required the entire message to be encoded before transmission would produce considerable delays. [2]

2.2.2 Underflow

It is possible that a sequence of symbols may occur in a message that causes the limits of the interval to repeatedly straddle 1/2. As this occurs, the most significant bits of the registers fail to converge, and less and less precision is available to calculate the effects of subsequent symbols. Ultimately, a situation may arise where

$$\text{HIGH} = 1000000000000000$$

$$\text{LOW} = 0111111111111111$$

and no further bits are available for calculations. In order to ensure that the message can be encoded and decoded correctly, this situation must be prevented.

To do this, the encoder examines the most significant two bits of the buffers. If these are 10 for HIGH and 01 for LOW, a counter x is incremented, the second most significant bit is deleted, and the bits below it are shifted up one place. The least significant bits of HIGH and LOW are filled with 1 and 0 respectively.

When the most significant bits of HIGH and LOW converge to a value b, this value is output followed by x instances of $\bar{b}$, and x is reset to zero.

This process may be considered in the following terms. When the most significant bits of HIGH and LOW are 10 and 01 respectively, we know that the ends of the interval lie in the range $[1/4, 3/4)$. We can therefore discard the subintervals $[0, 1/4)$ and $[3/4, 1)$, and consider the range $[1/4, 3/4)$ with greater precision. If, when the most significant bit eventually does converge, it converges to 0, we know that the interval lies in the range $[1/4, 1/2)$ and so the next bit to be sent must be 1. If the most significant bit converges to 1, we know that the interval lies in the range $[1/2, 3/4)$, so the next bit to be sent must be 0.

In the decoder, the examination of the two most significant bits of HIGH and LOW, and the shifting of bits to prevent loss of precision, are carried out in the same way as in the encoder, but there is no need to keep count of the discarded bits. [3]

2.2.3 Overflow

In practice, the cumulative probabilities c_k of the symbols are stored as integers, as floating point arithmetic is harder to implement and slower to execute. Furthermore, use of floating point numbers in numerical calculations can lead to loss of precision. The cumulative probabilities are represented as rational numbers, with the numerator f_k and the denominator f_{n-1} (where n is the

number of possible symbols). Therefore $f_k = c_k f_{n-1}$. To ensure that calculations are carried out to the maximum possible precision, when scaling operations are carried out, multiplications will be carried out before divisions. Unfortunately, an intermediate result may exceed the maximum value that can be stored in the registers. To avoid this, products are calculated in a register which is big enough to hold a value of at least MAX $\times f_{n-1}$, where MAX is the maximum value that can be stored in HIGH or LOW. The results of divisions are transferred back to the relevant registers afterwards. This must be implemented in both the encoder and the decoder.

2.2.4 Termination

It is necessary to provide a mechanism to signal to the decoder when the message is complete, so as to avoid the decoder appending random symbols to the message. There are three mechanisms that can be used to achieve this. The first is to signal in advance the size of the decoded message, so that the decoder can terminate when the required number of symbols have been decoded. This requires the symbols to be counted in advance of coding, which may introduce a small delay into the coding. A second alternative is to send messages composed of a fixed number of decoded symbols. The decoder can then decode the required number of symbols without need for it to be counted and sent in advance. For video coding applications, such as Dirac, this approach would be suitable in an error-free channel. However, as Dirac is likely to operate in errored channels, the number of coded bits is also transmitted, so that errors in the bit stream do not cause the decoder to run over the end of the frame. Finally, a special end of file (EOF) character may be encoded and sent at the end of the data. The decoder will terminate when this character is decoded. This is useful in applications where the length of the message is not known in advance, such as text encoding, but the EOF character, due to its low probability of occurrence, will be relatively expensive to send.

Whatever method is used to signal the end of the message, it is necessary to ensure that the last symbol of the message is completely sent, and can be decoded correctly.

2.3 The advantages of binarisation

So far we have considered a symbol alphabet of arbitrary size. If, however, the symbol alphabet is restricted to the binary symbols 0 and 1, we find that we need only store one probability value $p_0 = c_0$, as by definition $c_1 = 1$. Therefore, in the case where 0 is to be encoded,

$$\text{LOW}' = \text{LOW} + c_{-1}(\text{HIGH} - \text{LOW} + 0000000000000001) = \text{LOW}$$

since $c_{-1} = 0$

$$\text{HIGH}' = \text{LOW} + c_0(\text{HIGH} - \text{LOW} + 0000000000000001) - 0000000000000001$$

and where 1 is to be encoded,

$$\text{LOW}' = \text{LOW} + c_0(\text{HIGH} - \text{LOW} + 0000000000000001)$$

$$\text{HIGH}' = \text{LOW} + c_1(\text{HIGH} - \text{LOW} + 0000000000000001) - 0000000000000001 = \text{HIGH}$$

.

Therefore in each case, one of the registers is unaltered and only one calculation need be carried out. Furthermore, it is almost the same calculation in each case. The use of a binary symbol alphabet therefore greatly reduces the complexity of the encoder and decoder. The data to be encoded is therefore translated into a binary symbol alphabet prior to encoding. [4]

2.4 Adaptive probabilities

In the preceding discussion, we have assumed that the symbol probabilities are known in advance. This would be applicable in some circumstances, such as coding text in a known language, but in general it will not be the case. It is therefore necessary for the encoder to learn the symbol probabilities from the actual data. For a non-real time coder, it is possible to count symbol probabilities in the entire message beforehand and send them to the decoder, but this would introduce a delay that would be unacceptable in a real-time application. There would also be an overhead in the transmission of the symbol probabilities. An alternative, therefore is to encode each symbol using a statistical model based on previous symbol counts.

For the binary arithmetic coder described above, the symbol counts may be initialised as $f_0 = 1$, $f_1 = 2$. Whenever a symbol is encoded, f_1 is incremented, and whenever the symbol is 0, f_0 is also incremented. The decoder updates its probabilities accordingly on decoding the symbol. The difference between the probabilities used in the encoder and the actual probabilities is

$$\frac{1 - 2p_0}{f_1} = \frac{1 - 2p_0}{N + 2}$$

where N is the number of symbols that have been decoded. The probability therefore converges towards the true value as more symbols are encoded. There is some overhead from the initial inaccuracy of the probabilities, but this has been shown to be small. [4]

As the number of symbols encoded increases, so the ability of the coder to adapt to the symbol counts decreases. This effect may be reduced by periodically refreshing the symbol counts, for example by halving all symbol counts whenever f_1 reaches some maximum value. This does however mean that the speed with which the encoder adapts to the symbol statistics will vary discontinuously.

2.5 Context modeling

So far, we have assumed that the probabilities of successive symbols are independent. In many messages, this will not be the case. For example, if we were coding English text and encountered the symbol "q", we would know almost with certainty that the next symbol would be "u". Making use of this knowledge, we would be able to encode the symbol much more efficiently than would otherwise be the case.

More efficient arithmetic coding can therefore be achieved by maintaining a conditional probability model. This employs separate symbol counts for a number of different *contexts*, and selects the one appropriate for encoding each symbol according to the context in which it occurs.

2.6 Problems specific to hardware implementation

We wish to develop a dedicated hardware implementation of the Dirac codec. This would greatly speed coding and decoding and would enable Dirac codecs to be embedded in a variety of devices, such as cameras and PCs. It is difficult to perform division in hardware, unless it is by powers of two, in which case the operation can be reduced to a bitshift. Several possible methods may be used to address this problem.

One possible approach is to store new symbol counts in a separate register, and add these to the existing symbol counts only when they make up the total to a power of two. This, however, would mean that adaptation took place discontinuously.

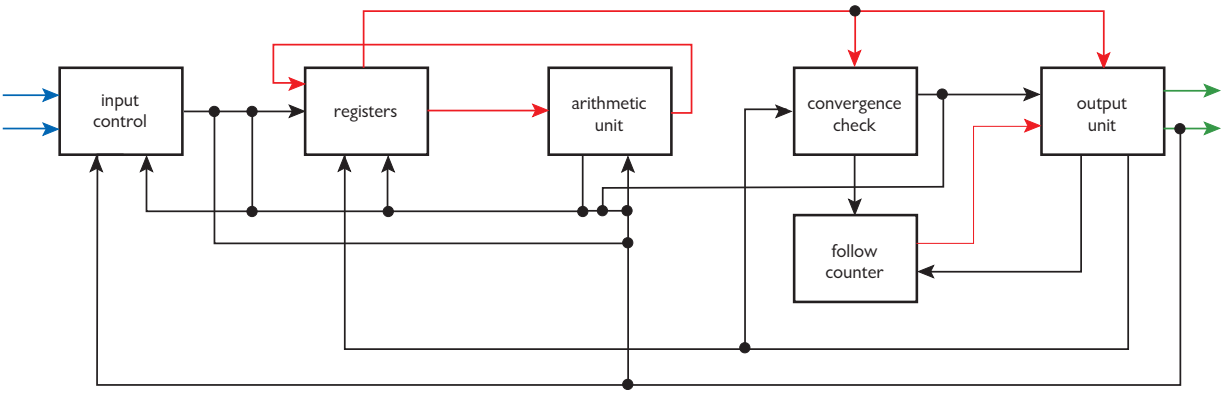

Figure 1: Encoder architecture. Inputs are blue, control signals black, arithmetic data red and outputs green

Another possibility is to find some constant Q such that

$$\left| \frac{1}{Qf_1} - 2^{-z} \right| < \epsilon$$

for some integer z, and then make the approximation

$$p_0 = \frac{f_0}{f_1} \approx \frac{Qf_0}{2^z}$$

As long as $\epsilon < \frac{1}{\text{MAX}}$, there would not be significant loss of precision by this method. Suitable values of Q for each value of f_1 may be stored in a look-up table.

A third possible method is to keep f_1 at a constant value of 1024, and store the last 1024 symbols encoded for each context. When a new symbol is encoded, it is compared to the symbol that was encoded 1024 symbols ago, and f_0 is updated accordingly. This coder will initially learn more slowly than a those previously described, but it its rate of learning will be constant.

3 Encoder architecture

The arithmetic coder has four inputs, ENABLE, DATA_IN, RESET, and CLOCK, and two outputs, SENDING and DATA_OUT.

When ENABLE is 1, new data can be read at DATA_IN and processed by the encoder. The RESET signal is used to restore the encoder's internal state to initial conditions. The encoder's operations are synchronised by means of the CLOCK signal. When encoded data is available for output, SENDING is set to 1, and the encoded data can be read from DATA_OUT — that is SENDING provides an enable signal for further stages of processing. The overall architecture of the coder is shown in figure 1.

3.1 The components of the encoder

The encoder consists of the following components

- an INPUT_CONTROL, which stores incoming data until the encoder is ready to code it.
- three LIMIT_REGISTERS, named LOW, DIFFERENCE and HIGH. These store the values used in the encoder calculations,

- a CONVERGENCE_CHECK module,
- an ARITHMETIC_UNIT, which performs the calculations used by the encoder,
- an OUTPUT_UNIT, which performs output when necessary,
- a FOLLOW_COUNTER, which implements the "bits plus follow" mechanism used to prevent underflows, and
- additional logic to coordinate the operation of these components.

These components are described in greater detail below.

3.1.1 INPUT_CONTROL

The INPUT_CONTROL has six 1 bit wide inputs, ENABLE, DATA_IN, BUFFER_CONTROL, DEMAND, RESET, and CLOCK. It has two 1 bit wide outputs, SENDING and DATA_OUT.

Internally, any data is stored in a FIFO. If this FIFO is EMPTY and BUFFER_CONTROL is 0, the values of ENABLE and DATA_IN are passed directly to SENDING and DATA_OUT respectively. Otherwise, when ENABLE is 1, the value of DATA_IN is stored in the FIFO. If there is data in the FIFO, the value of DATA_OUT is read from the FIFO when DEMAND is 1, and the value of DEMAND is passed to SENDING. RESET is used to set the INPUT_CONTROL back to an EMPTY state.

The FIFO is parameterised by an integer generic RANK, which has a default value of 8. It has five 1 bit wide inputs, WRITE_ENABLE, DATA_IN, READ_ENABLE, RESET and CLOCK, and two 1 bit wide outputs, DATA_OUT and EMPTY. When WRITE_ENABLE is 1, DATA_IN is stored. If EMPTY is 1, it becomes 0. The value of the oldest unread data is present at DATA_OUT. When READ_ENABLE is 1, this value will be discarded on the following clock cycle. If this leaves the FIFO EMPTY, EMPTY will be set to 1. If RESET is 1, the FIFO will be reset to an empty state. The FIFO has a capacity of 2^{RANK} bits; therefore, the default capacity of the FIFO is 256 bits.

3.1.2 LIMIT_REGISTER

A LIMIT_REGISTER stores a 16 bit value. It is specified with a generic value, CONST, which is determined at instantiation. This is 0 for LOW and 1 for HIGH and DIFFERENCE. It has a 16 bit wide input LOAD, and five further 1 bit wide inputs, SET_VALUE, SHIFT_ALL, SHIFT_MOST, RESET and CLOCK. It has one 16 bit wide OUTPUT.

When SET_VALUE is 1, the value of LOAD is stored in the LIMIT_REGISTER. When SHIFT_ALL is 1, then all bits of the value are shifted up one place, the most significant bit is discarded, and the least significant bit set to CONST. When SHIFT_MOST is 1, then the second most significant bit is discarded, the remaining bits are shifted up one place, and the least significant bit is set to CONST. When RESET is 1, all the bits stored in the register are set to CONST. CLOCK is used to synchonise the operations of the component. The value stored in the LIMIT_REGISTER can be read from OUTPUT. The value is stored and manipulated by 16 STORE_BLOCKs.

A STORE_BLOCK has five inputs, LOAD_IN, SHIFT_IN, ENABLE, SHIFT, and CLK. It has one OUTPUT. As long as ENABLE is 0, OUTPUT will be constant. When ENABLE is 1, OUTPUT will be set to a new value at the next rising clock edge detected at CLK. If SHIFT is 1, this value will be the value of SHIFT_IN, otherwise it will be LOAD_IN.

Each STORE_BLOCK's LOAD_IN value is the corresponding bit of the LIMIT_REGISTER's LOAD input. If RESET is 0, CONST if RESET is 1. Its SHIFT_IN input is connected to the OUTPUT of the STORE_BLOCK representing the previous bit. For the least significant bit, it is

set at CONST. ENABLE is (SHIFT_ALL or SET_VALUE or RESET) for the most significant bit, (SHIFT_ALL or SHIFT_MOST or SET_VALUE or RESET) for the remainder. CLK is connected to the LIMIT_REGISTER's CLOCK input. Each STORE_BLOCK's OUTPUT is connected to the relevant bit of the LIMIT_REGISTER's OUTPUT.

3.1.3 CONVERGENCE_CHECK

The CONVERGENCE_CHECK module has five inputs, HIGH_MSB, LOW_MSB, HIGH_SECONDBIT, LOW_SECONDBIT and CHECK. It has two outputs, TRIGGER_OUTPUT and TRIGGER_FOLLOW.

When CHECK is 0, both outputs will be 0. When CHECK is 1, TRIGGER_OUTPUT will be 1 if the inputs HIGH_MSB and LOW_MSB are the same, 0 if they are different. TRIGGER_FOLLOW will be 1 if HIGH_MSB and LOW_MSB are different, HIGH_SECONDBIT is 0 and LOW_SECONDBIT is 1.

3.1.4 ARITHMETIC_UNIT

The ARITHMETIC_UNIT has two 16 bit wide inputs, DIFFERENCE and LOW, one 10 bit wide input, PROB, and two 1 bit wide inputs, ENABLE and CLOCK. It has four 16 bit wide outputs, DIFFERENCE_OUT0, DIFFERENCE_OUT1, RESULT_OUT0 and RESULT_OUT1, and one 1 bit wide output, DATA_LOAD.

ENABLE is used to indicate that the values of DIFFERENCE and LOW are stable and may be used for calculation. DATA_LOAD indicates that a calculation is complete, and the values of the numerical outputs are stable.

DIFFERENCE is expanded to 17 bits by adding a leading zero, and 1 is added. This signal is designated DIFFERENCE2. This is multiplied by PROB in a clocked multiplier, giving a 27 bit wide result, and the result truncated to the 17 most significant bits. This signal is designated DIFFERENCE2. LOW is delayed by one clock cycle and has a leading zero added to form LOW2. DIFFERENCE2 and LOW2 are added together. The leading bit of one copy of this is discarded to give RESULT_OUT0. 1 is subtracted from another copy, and the leading bit discarded to give RESULT_OUT1.

1 is subtracted from DIFFERENCE2, the most significant bit discarded, and the result delayed by 1 clock cycle to give DIFFERENCE_OUT0. A copy of DIFFERENCE2 is delayed by 1 clock cycle, DIFFERENCE2 subtracted from it, and the most significant bit discarded to give DIFFERENCE_OUT1.

The delays are used to ensure that all arithmetic operations take place on stable signals.

ENABLE is passed through two D_TYPEs in series, and the outputs of each of them combined in an AND gate to form DATA_LOAD. This ensures that the inputs have been stable for two clock cycles before the outputs are read.

3.1.5 OUTPUT_UNIT

The output unit has four inputs, ENABLE, DATA, FOLLOW, and CLOCK. It has four outputs, SENDING, DATA_OUT, FOLLOW_COUNTER_TEST and SHIFT.

The value of SENDING is (ENABLE and not RESET and not SHIFT). DATA_OUT is set to ((DATA XOR FOLLOW) and ENABLE). FOLLOW_COUNTER_TEST is set to ENABLE delayed by one clock cycle. SHIFT is set to (FOLLOW_COUNTER_TEST and not FOLLOW).

3.1.6 FOLLOW_COUNTER

The FOLLOW_COUNTER has four inputs, INCREMENT, TEST, RESET, and CLOCK. It has one OUTPUT. The internal state of the FOLLOW_COUNTER represents an 8 bit number. Whenever

INCREMENT is 1, this number is incremented. Whenever TEST is 1, OUTPUT is 1 if the number is non-zero, and the counter is decremented for as many clock cycles as it takes to reach zero.

The number is stored in the follow counter in chain of eight COUNT_UNITs. A COUNT_UNIT has four inputs, INCREMENT, DECREMENT, RESET and CLOCK. It has three outputs, OUTPUT, INCREMENT_CARRY and DECREMENT_CARRY. If INCREMENT, DECREMENT and RESET are all 0, the value of OUTPUT will remain unchanged at the next rising clock edge. If INCREMENT or DECREMENT is 1, OUTPUT will be inverted on the next rising clock edge. If RESET is 1, OUTPUT will be set to 0 on the next rising clock edge. INCREMENT_CARRY is defined as (INCREMENT and OUTPUT), and DECREMENT_CARRY is defined as (DECREMENT and not OUTPUT).

The COUNT_UNITs representing the seven most significant bits each take their INCREMENT and DECREMENT inputs from the INCREMENT_CARRY and DECREMENT_CARRY outputs of the COUNT_UNIT representing the preceding bit respectively. The COUNT_UNIT representing the least significant bit takes its INCREMENT input from the INCREMENT input of the FOLLOW_COUNTER. All the COUNT_UNITs receive their RESET and CLOCK signals from those of the FOLLOW_COUNTER. The OUTPUTs of all the COUNT_UNITs are combined in an or-gate to produce a signal NONZERO. The OUTPUT of the FOLLOW_COUNTER is defined as (TEST and NONZERO), and is fed back to the DECREMENT input of the COUNT_UNIT representing the least significant bit.

3.2 Combining the components

All components in the arithmetic coder that have a RESET or CLOCK input receive this directly from the coder's RESET and CLOCK inputs respectively.

The INPUT_CONTROL receives its ENABLE and DATA_IN inputs from those of the coder. Its BUFFER_CONTROL input is 1 when the OUTPUT_UNIT's SENDING output is 1 or the ARITHMETIC_UNIT's DATA_LOAD output is 0. Its DEMAND input is 1 when the ARITHMETIC_UNIT's DATA_LOAD output is 1.

The LIMIT_REGISTER LOW receives its LOAD input from the ARITHMETIC_UNIT's RESULT_OUT1 output. It's SET_VALUE input is 1 when the INPUT_CONTROL's SENDING output, the ARITHMETIC_UNIT's DATA_LOAD output and the INPUT_CONTROL's DATA_OUT output are all 1. Its SHIFT_ALL input is taken from the OUTPUT_UNIT's SHIFT output. Its SHIFT_MOST output is taken from the CONVERGENCE_CHECK's TRIGGER_FOLLOW output. Its CONST value is 0.

The LIMIT_REGISTER HIGH has similar inputs to LOW, except that its LOAD input is taken from the ARITHMETIC_UNIT's RESULT_OUT0 output. SET_VALUE input is 1 when the INPUT_CONTROL's SENDING output and the ARITHMETIC_UNIT's DATA_LOAD output are 1 and the INPUT_CONTROL's DATA_OUT input is 0. Its CONST value is 1.

The LIMIT_REGISTER DIFFERENCE takes its LOAD input from the ARITHMETIC_UNIT's DIFFERENCE_OUT0 output if the INPUT_CONTROL's DATA_OUT output is 1, and from the ARITHMETIC_UNIT's DIFFERENCE_OUT1 output if it is 1. Its SET_VALUE input is taken from the ARITHMETIC_UNIT's DATA_LOAD output. Its SHIFT_ALL input is 1 when the OUTPUT_UNIT's SHIFT output is 1 or the CONVERGENCE_CHECK's TRIGGER_FOLLOW output is 1. Its SHIFT_MOST input is 0. Its CONST value is 1.

The CONVERGENCE_CHECK's HIGH_MSB input is the most significant bit of HIGH's OUTPUT. Its LOW_MSB input is the most significant bit of LOW's OUTPUT value. HIGH_SECONDBIT and LOW_SECONDBIT are the second most significant bits of HIGH and LOW's OUTPUTs respectively. CHECK becomes 1 one clock cycle after the OUTPUT_UNIT's SHIFT output becomes 1 or the CONVERGENCE_CHECK's TRIGGER_FOLLOW output becomes 1 or the INPUT_CONTROL's SENDING output and the ARITHMETIC_UNIT's DATA_LOAD output are both 1.

The ARITHMETIC_UNIT's DIFFERENCE input is taken from DIFFERENCE's OUTPUT. Its PROB

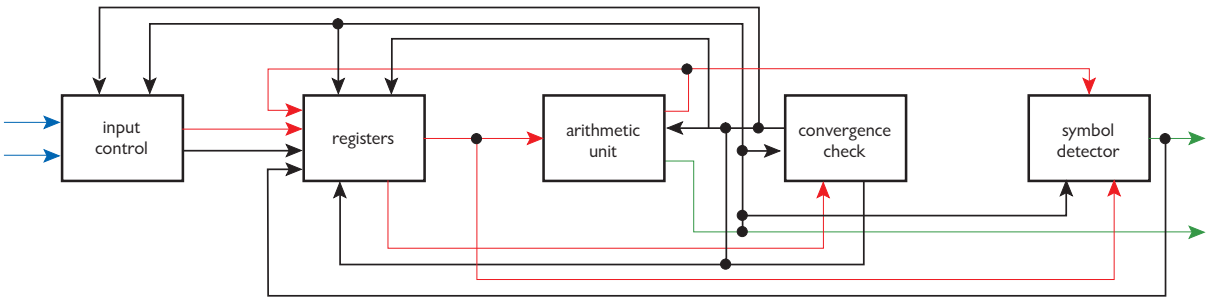

Figure 2: Decoder architecture. Inputs are blue, control signals black, arithmetic data red and outputs green

input is currently a constant. Its LOW input is taken from LOW's OUTPUT. Its ENABLE input is 1 when all the following conditions are met

- the OUTPUT_UNIT's SENDING output is 0
- the CONVERGENCE_CHECK's outputs are both 0
- either of the ARITHMETIC_UNIT's DATA_LOAD output or the INPUT_CONTROL's SENDING output are 0.

The OUTPUT_UNIT's ENABLE input is taken from the CONVERGENCE_CHECK's TRIGGER_OUTPUT output. Its DATA input is taken from the most significant bit of HIGH's output. Its FOLLOW input is taken from the FOLLOW_COUNTER's OUTPUT.

The FOLLOW_COUNTER's INCREMENT input is taken from the CONVERGENCE_CHECK's TRIGGER_FOLLOW output. Its TEST input is taken from the OUTPUT_UNIT's FOLLOW_COUNTER_TEST input.

The coder's SENDING output and its DATA_OUT output are taken from the similarly named outputs of the OUTPUT_UNIT.

4 Decoder architecture

The decoder has the same interface as the the encoder. The overall architecture of the decoder is shown in figure 2.

4.1 The components of the decoder

The components of the decoder are similar to those of the encoder. However, rather than LIMIT_REGISTERs, the decoder's working data is stored in four STORAGE_REGISTERs, HIGH, LOW, CURRENT and DIFFERENCE. There is no FOLLOW_COUNTER or OUTPUT_UNIT. There is, however, a SYMBOL_DETECTOR, which generates the decoded output.

4.1.1 STORAGE_REGISTER

A STORAGE_REGISTER has a 16 bit wide input, LOAD. It has a further six 1-bit wide inputs, SHIFT_IN, SET_VALUE, SHIFT_ALL, SHIFT_MOST, RESET and CLOCK. It has one 16 bit wide OUTPUT.

The STORAGE_REGISTER is similar to a LIMIT_REGISTER, except that when SHIFT_ALL or SHIFT_MOST is triggered, the least significant bit is filled with the value of SHIFT_IN, and when RESET is triggered all values are filled with zeros.

4.1.2 SYMBOL_DETECTOR

The SYMBOL_DETECTOR has a 1 bit input, ENABLE, two 16 bit wide inputs, DATA_IN and THRESHOLD, and a 1 bit output, DATA_OUT. When ENABLE is 1 and DATA_IN is greater than or equal to THRESHOLD, DATA_OUT is 1. Otherwise it is 0.

4.2 Combining the components

All components that have a CLOCK or RESET input take this from the corresponding input of the decoder.

The INPUT_CONTROL takes its ENABLE and DATA_IN inputs from those of the decoder. Its BUFFER_CONTROL input is 1 if the ARITHMETIC_UNIT's DATA_LOAD output is 1 or the CONVERGENCE_CHECK's TRIGGER_OUTPUT and TRIGGER_FOLLOW outputs are both 0. Its DEMAND input 1 if the either of the CONVERGENCE_CHECK's outputs is 1

The STORAGE_REGISTER LOW takes its LOAD value from the ARITHMETIC_UNIT's RESULT_OUT1 output. Its SHIFT_IN value is 0. Its SET_VALUE is 1 when the SYMBOL_DETECTOR's DATA_OUT is 1 and the ARITHMETIC_UNIT's DATA_LOAD is 1. Its SHIFT_ALL input is 1 when the CONVERGENCE_CHECK's TRIGGER_OUTPUT output and the INPUT_CONTROL's SENDING output are both 1. Its SHIFT_MOST value is 1 when the CONVERGENCE_CHECK's TRIGGER_FOLLOW output and the INPUT_CONTROL's SENDING output are both 1.

The STORAGE_REGISTER HIGH has similar inputs to those of LOW, except that it takes its LOAD value from the ARITHMETIC_UNIT's RESULT_OUT0 output, its SHIFT_IN input is 1, and its SET_VALUE is 1 when the SYMBOL_DETECTOR's DATA_OUT is 0 and the ARITHMETIC_UNIT's DATA_LOAD is 1.

The STORAGE_REGISTER DIFFERENCE's LOAD input is taken from the ARITHMETIC_UNIT's DIFFERENCE_OUT1 output when the SYMBOL_DETECTOR's DATA_OUT output is 1, and from the ARITHMETIC_UNIT's DIFFERENCE_OUT0 output when the SYMBOL_DETECTOR's DATA_OUT output is 0. Its SET_VALUE is taken from the ARITHMETIC_UNIT's DATA_LOAD output. Its SHIFT_IN input is 0. It's SHIFT_ALL input is 1 when the INPUT_CONTROL's SENDING output is 1 and either of the CONVERGENCE_CHECK's outputs is 1. Its SHIFT_MOST input is zero.

The STORAGE_REGISTER CURRENT's LOAD value is 0000000000000000. Its SHIFT_IN value is taken from the INPUT_CONTROL's DATA_OUT output. Its SET_VALUE input is 0. Its SHIFT_ALL and SHIFT_MOST inputs are the same as for LOW and HIGH.

The CONVERGENCE_CHECK's HIGH_MSB input is the most significant bit of HIGH's OUTPUT. LOW_MSB is the most significant bit of LOW's OUTPUT. HIGH_SECONDBIT and LOW_SECONDBIT are the second most significant bits of HIGH and LOW's OUTPUTs respectively. Its CHECK input is 1 when the ARITHMETIC_UNIT's DATA_LOAD output is zero.

The ARITHMETIC_UNIT's DIFFERENCE input is taken from DIFFERENCE's OUTPUT. Its PROB input is currently set to a CONSTant. Its LOW input is taken from LOW's OUTPUT. Its ENABLE input is 1 when its DATA_LOAD output and both the CONVERGENCE_CHECK's outputs are all zero.

The SYMBOL_DETECTOR's ENABLE input is taken from the ARITHMETIC_UNIT's DATA_LOAD output. Its DATA_IN input is taken from CURRENT's OUTPUT, and its THRESHOLD input is taken from the ARITHMETIC_UNIT's RESULT_OUT1 output.

The decoder's SENDING output is taken from the ARITHMETIC_UNIT's DATA_LOAD output, and its DATA_OUT output is taken from the SYMBOL_DETECTOR's DATA_OUT output.

5 Synthesis and simulation

The Xilinx Virtex II family is a class of FPGAs comprising 11 models ranging from 40K to 8M system gates. Their configurable components are organised as a regular array of configurable logic blocks, which provide combinatorial and synchronous logic functions, with dedicated routing; block and distributed RAM; 18×18 bit multipliers; and digital clock managers and global clock buffers, which allow clock speeds of up to 420MHz. Configurable logic blocks (CLBs) consist of four "slices", each of which contains two four-input function generators, carry logic, arithmetic logic gates, wide function multiplexors and two storage elements. The function generators can be programmed as four input lookup tables, distributed RAM, or 16 bit shift registers. The storage elements can be configured as D-type flip-flops or level sensitive latches. Within the CLB there are fast connections between slices, and each CLB has access to the general routing matrix of the FPGA. Block RAM is available in 18 Kbit blocks, with an 18×18 bit multiplier associated with each block. Digital clock managers are used to generate de-skewed clock signals and eliminate clock distribution delay. There are also programmable I/O blocks that enable a wide variety of I/O standards to be implemented. [5] As a guide to pricing, the XC2V1000, a middle-range FPGA with 1M system gates, retails for around US$430, for individual units.

An FPGA from the lower range of the Virtex II family, the Virtex II XC2V250cs144-6 FPGA, has 250K system gates, arranged in an array of 24×16 CLBs, with a total of 1536 slices, 24 multipliers, and 24 RAM blocks, giving a total of 432Kbits of block RAM. It has 8 DCMs, 12 global clock multiplexors, and 200 I/O pads. Synthesizing the arithmetic coder design for this FPGA gave the following device utilisation statistics.

- 405 slices used out of a possible 1536 (26%)
- 392 slice flip flops used out of a possible 3072 (12%)
- 532 four input LUTs used out of a possible 3072 (17%)
- 5 bonded IOBs used out of a possible 92 (5%)
- 1 18×18 bit multiplier used out of a possible 24 (4%)
- 1 digital clock manager used out of a possible 16 (6%)

The minimum clock period was 9.336 ns.

Synthesizing the decoder for the same target FPGA gave the following device utilisation statistics

- 419 slices (27%)
- 390 slice flip flops (12%)
- 542 four input LUTs (17%)
- 5 bonded IOBs (5%)
- 1 18×18 bit multiplier (4%)
- 1 digital clock manager (6%)

The minimum clock period was 8.119 ns.

It is therefore possible to run the encoder and decoder at a clock speed of 100MHz. As three clock cycles are needed to encode or decode a symbol (two to calculate the new values and one to load them into the registers), further clock cycles are required when data is to be output, data can therefore be encoded at a rate of 1 symbol every 4 clock cycles, or 25 Msymbol s^{-1}.

A 1Mbit pseudorandom data file was created with $f_0 = 912$, $f_1 = 1024$. This corresponds to an entropy of 0.498 bit $symbol^{-1}$. This was passed through the encoder. This was coded to 512539 bits, around 2% less than predicted from the source entropy.

By connecting the outputs of the encoder directly to the inputs of the encoder in a testbench, and saving the decoded output to a file, it was possible to confirm that the decoded output matched the original input.

5.1 Further work

This arithmetic coder/decoder only implements a fixed probability model. Further features are needed to implement the full functionality required by Dirac. These are

- conditional probabilities
- binarisation of multibit symbols
- probability adaptation.

The VHDL for the arithmetic coder and decoder described in this work can be found at *http://dirac.sourceforge.net/* and at *http://www.opencores.net/projects.cgi/dirac/overview*

6 Acknowledgments

Thanks to Marc Servais of Surrey University, whose review of the literature of Arithmetic Coding helped immensely with the compilation of this report.

References

[1] Glen G. Langton, Jr. An introduction to arithmetic coding. *IBM Journal of Research and Development*, 28(2), 1984.

[2] Frank Rubin. Arithmetic stream coding using fixed precision registers. *IEEE TRANSACTIONS ON INFORMATION THEORY*, IT-25(6), 1979.

[3] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6), 1987.

[4] Glen G. Langdon, Jr and Jorma Rissanen. Compression of black-white images with arithmetic coding. *IEEE Transactions on Communications*, COM-29(6), 1981.

[5] Xilinx Corporation. *Virtex-II Platform FPGAs: Complete Data Sheet*, Jun 2004.